Behavior Programming Language and Automated Code Generation for Agent Behavior Control

Thuc Vu
School of Computer Science
Carnegie Mellon University

Advisor: Professor Manuela Veloso

April 30, 2004

Contents

Abstract

Behavior-based agents are becoming increasingly used across a variety of implementation platforms. The common approach to building such agents involves implementing the behavior synchronization and management algorithms directly in the agent's programming environment. This process makes it hard, if not impossible, to share common components of a behavior architecture across different agent implementations. This lack of reuse also makes it cumbersome to experiment with different behavior architectures as it forces users to manipulate native code directly, e.g. C++ or Java. In this thesis, we provide a high-level behavior-centric programming language and an automated code generation system which together overcome these issues and facilitate the process of implementing and experimenting with different behavior architectures. The language is specifically designed to allow clear and precise descriptions of a behavior hierarchy, and can be automatically translated by our generator into C++ code. Once compiled, this C++ code yields an executable that directs the execution of behaviors in the agent's sense-plan-act cycle. We have tested our framework with different platforms, including both software and robot agents, with various behavior architectures. We experienced the advantages of defining an agent by directly reasoning at the behavior architecture level followed by the automatic native code generation.

Chapter 1

Introduction

Agent and multi-agent systems play a very important role in many different areas including robotics, auctions, and web applications. They provide an elegant and efficient solution for the control problems in those areas. There have been many research projects on different agent behavior architectures in order to find the optimal solutions for certain applications. Most of the architectures however may be applicable across platforms and can be reused to solve a variety of different problems.

1.1 Motivation

The common current approach to applying agent and multi-agent systems to a new platform is to implement the entire architecture even though there can be a large overlapping with a previously implemented architecture framework such as the behavior management code. For an example, the "search and rescue" behavior for on-ground vehicles should be very similar to that of underwater ones; the only difference is the underlying mechanism for actuating the vehicles. It can be redundant to reimplement the entire architecture. Moreover, one usually needs to experiment with different behavior architectures in order to find the optimal one for the current platform. Within the current approach, one will have to manipulate the native code of the implementation which is very much time consuming and error prone.

1.2 Example of a Behavior-based System

A well-known example of a behavior-based multi-agent system is robot soccer in which several teams of robots compete against each other. One particular team is the CMPack'03 from Carnegie Mellon University [6]. The CMPack'03 is composed of several autonomous legged soccer robots as in Figure 1.1. Each robot has its own vision and motion system. The vision system will provide the robot the perception of the objects and the position of itself on the field. The motion system allows the robot to perform certain actions in the environment such as walking in a particular direction, or kicking the ball using a particular type of kick.

The CMPack'03 also has a behavior system to coordinate the robots towards achieving a specific goal. Each robot is considered as an agent that is capable of making individual autonomous decisions contributing towards the team goals. Each behavior is represented as a finite state machine which may contain encapsulated state machine. The transitions between the states are dependent on the condition of the world and the robots themselves. Those transitions can be arbitrarily complex.

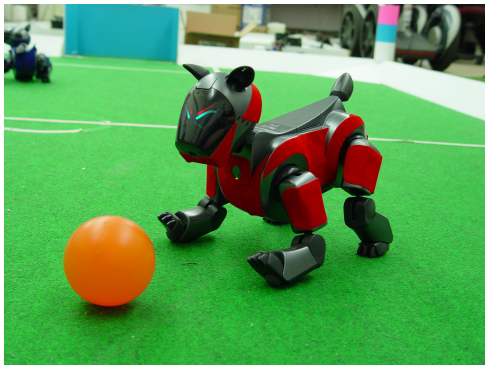

Figure 1.1: A picture of the AIBO.

Each agent is assigned a specific role. The agents playing offense can be assigned one of the three separate roles: primary attacker, offensive supporter and defensive supporter. There is only one agent in the role of primary attacker at one time. This is maintained through using a token to ensure mutual exclusion. The primary attacker will move directly to the ball and attempt to score while the supporting attackers will use a potential field to position themselves.

1.3 Our Previous Work in Agent Programming

In MONAD [7], one of our earlier works, we develop an intuitive and flexible scripting language which enables designers to easily build a team in the form of an augmented behavior hierarchy. This behavior hierarchy facilitates modification of various parameters of a team, such as the structure of the hierarchy itself and the negotiation protocols to be used by the agents during synchronized execution of the hierarchy. In order to execute the team structures specified by MONAD scripts, the MONAD architecture includes a run-time distributed behavior-based control engine called SCORE (Synchronized CoORdination Engine) which is able to execute any team design specified by the MONAD scripting language. SCORE synchronizes the execution of behaviors across multiple agents, drawing on a user-designed reusable library of negotiation protocols that support different team-control designs.

In more details, the MONAD system is composed of several key components as in Figure 1.2 which work together to provide a flexible framework for multi-agent programming. The components provided by the designer include both offline script and code, in the form of the team program, team description file, arbitration execution code, and behavior execution code. The team program and description file are both written in a format specified by the MONAD architecture, while the arbitration and behavior execution must be written and compiled to native code. Given these inputs, the MONAD architecture provides synchronized execution of the team program through SCORE which acts as a distributed coordination system for the entire team. Each agent on a team runs an identical copy of SCORE and initializes it with the same team program and a team description file, along with an index which uniquely identifies to SCORE which agent it is controlling.

Using the scripting language greatly reduces our time and effort in implementing a new behavior architecture as well as experimenting with different parameters of the architecture. However, the scripting language has a restricted expressiveness. It can only represent behavior hierarchy in the form of a Direct Acyclic Graph. Furthermore, SCORE is designed specifically for the MONAD platform. One will have to modify and incorporate SCORE into another platform before using it and this may be infeasible due to constraints of resources such as processing power, or storage.

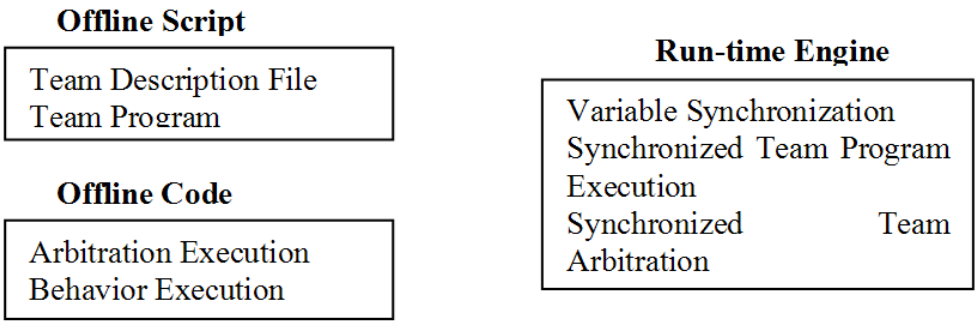

Figure 1.2: An overview of the MONAD system.

1.4 Our Novel Framework for Agent Programming

We conjecture that it will be beneficial to generalize the scripting language and the behavior control routines in MONAD (SCORE) to platform-independent and automate the process of generating these behavior control routines for a specific platform. Through this, we are separating the process of designing agent behavior architecture for high-level behaviors and incorporating the architecture to a given agent platform with implemented platform-dependent atomic behaviors. A designer, once specialized in one agent architecture, can reapply the architecture in existing agent platforms with minimum additional effort.

Towards that goal, this thesis introduces a novel framework composed of an intuitive and flexible high-level behavior-based language (HLBL) that can be used to describe easily the architecture for an agent to facilitate the designing process, and a code generation system (B2C) to automate the integrating process. To our best knowledge, this work contributes the first platform-independent behavior-based programming language that is combined with an automated translation to native code.

Using the given high-level language, the designer can build an agent in the form of an augmented behavior hierarchy. This hierarchy will be specified in the behavior description file, which will indicate how the execution of behaviors should unfold at runtime. The given behaviors can be combined in different ways by the structure of the behavior hierarchy, leading to a variety of patterns of behaviors executed by the agent. This system allows changes in the behavior hierarchy to be made at an intuitive level without the need to change native code. Further-

more, when applying the architecture to a different platform, the designer can usually reuse part of the behavior description file. Thus the framework can significantly facilitate the process of developing and experimenting an agent behavior architecture.

Given the behavior hierarchy description file as input, the code generation system will generate the correspondent C++ code. After being translated, the behavior hierarchy is output as a ”.cpp” file where each behavior in the hierarchy is a function in the .cpp file. The root of the hierarchy acts as the entry point to the agent control routines. The user then compiles and links this file with the rest of the project and at some point in the main program calls the function representing the root behavior in the hierarchy. This begins automated execution and the translated framework will handle running the behaviors from this point onwards (Figure 1.3).

Since the behavior description in HLBL is independent of the platform, we were able to applied the framework of HLBL and B2C to various agent platforms, both robot and software agents. In particular, we have developed a behavior control for the AIBO (Figure 2.3 and Figure 2.4), and two software agents in simulated environment (Section 4.1 and Section 4.2) in a very short time. The framework also allowed us to quickly set up different behavior architectures for experimentation in the pursuit of the optimal one. This demonstrates the advantages of using the framework in agent programming.

1.5 Reading Guidelines

After Chapter 1 of introduction, Chapter 2 and 3 present two main components of the framework respectively: the High-Level Behavior Language (HLBL) for designing agent behavior architecture and the Automated Code Generation (B2C) system for integrating the architecture to an existing framework. Chapter 4 shows several applications of HLBL and B2C as well as the advantages through using the new framework. Chapter 5 concludes the thesis and briefly discusses about possible extension of the framework.

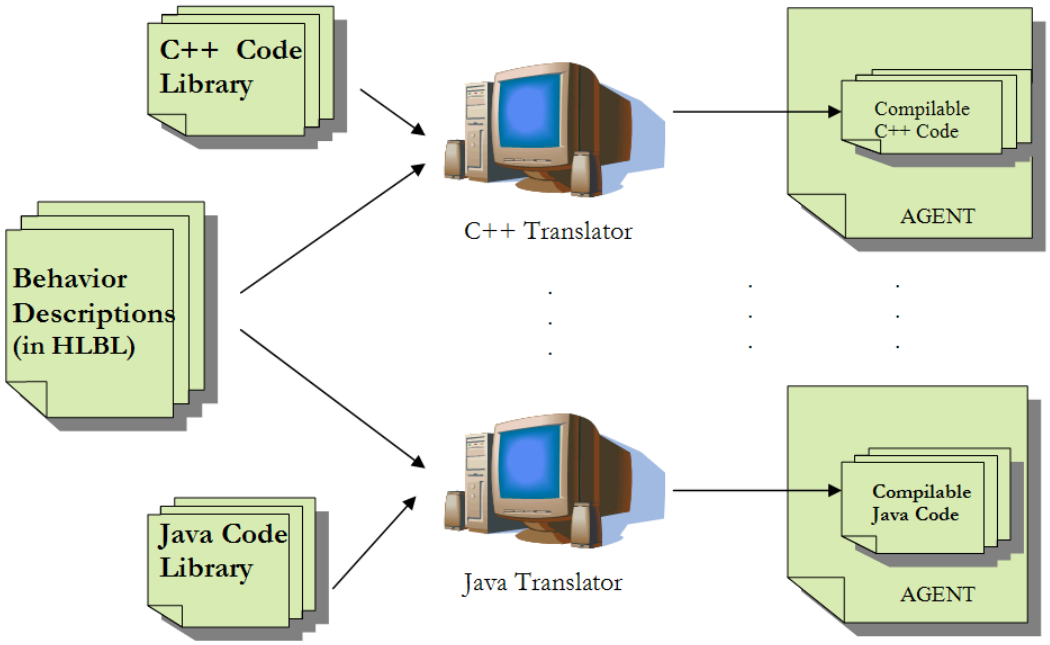

Figure 1.3: An overview of the connection between HLBL and the Translators.

Chapter 2

High-Level Behavior Language

High-level behavior language (HLBL) is a very flexible yet intuitive language which a designer can use to implement the behavior architecture for the agents. It is also platform−independent so that an agent architecture implemented for one specific platform in HLBL can be reused in the solution for another platform. HLBL allows the designer to focus more on the structure of the behaviors without being burdened with the tedious implementation details. It also facilitates process of experimenting with a variety of behavior architectures in finding the optimal one for a given problem.

2.1 Representation and Execution

In HLBL, the behavior control routines for an agent have the representation of a behavior hierarchy. HLBL is flexible enough to also represent any finite state machine in which each state is correspondent to one behavior. Thus the complexity of agent's behaviors is not restricted to any form.

Starting from the root of the hierarchy which serves as the entry point, the agent will execute the behavior and make the appropriate transitions based on the conditions of itself and the environment. Following is an example of a behavior hierarchy with 7 behaviors from B0 to B6 as shown in Figure 2.1. The labels SC, EC, and A denotes the set of starting, ending conditions and the action for each behavior respectively. Ci stands for the conditions for the agent to make the transition to behavior Bi. All those fields will be discussed further in the next paragraphs.

Each behavior is associated with a set of *starting conditions* and a set of *end-*

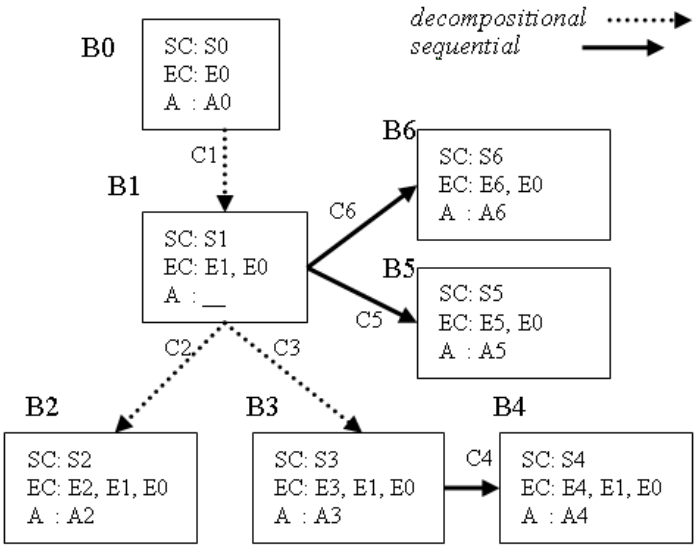

Figure 2.1: A behavior hierarchy

ing conditions. The starting conditions determine when a behavior is applicable and the ending conditions determine when the agent should stop executing the current behavior. Each behavior is also associated with a goal that the agent tries to accomplish through the behavior. This goal is included in the set of ending conditions. As an example, the ending conditions of B2 are E2, E1, and E0. If any of those conditions evaluates to true, the agent will stop executing B2. The agent will then make the appropriate transition.

There are two types of transitions from one behavior: *sequential* transition to a following behavior and *decompositional* to a child one. The child-behaviors represent the alternative ways to accomplish the goal of the parent. The following behaviors share the same parent with the followed if there is any and they contains the sequence of goals that the agent needs to accomplish in the specified order to achieve the goal of the parent behavior. In figure 1, B1 has two direct children B2, and B3 and one indirect child B4 since B4 follows B3. If the agent chooses to execute B3 instead of B2, it will have to also accomplish B4 in order to accomplish B1.

When making a decompositional transition, the agent is considered as still trying to achieve the goal of the parent behavior. Thus, the set of ending conditions

of a behavior contains as a subset the union of the ending conditions of its ancestors. In contrast, the agent has to accomplish the goal of the current behavior first before it can make a sequential transition. For an example, the ending conditions of B3 is E3, E1, and E0 since B0 and B1 are the ancestors of B3. Once the goal of a behavior is accomplished, if there is no following behavior, the agent will make the transition back to the parent of the current behavior.

There can be an *action* for each behavior. The action can be any behavior but preferably an atomic execution behavior. Atomic behavior is the behavior at the lowest level of the behavior hierarchy and it is not composed of any other behaviors. The action will be carried out and completed for every sense-plan-act cycle that the agent stays in the current behavior until either the ending conditions meet or the agent can make a transition to the child-behavior. All the behaviors having no children must have an action. In figure 1, B1 does not have an action but B2 to B6 must have an action because they do not have any children.

Once the ending conditions of a behavior are met, the agent will stop executing the current behavior. If the goal of an ancestor of the current behavior has been accomplished or if the behavior does not have any following, the agent will return to the parent behavior. Otherwise it will make the transition to the following behavior.

A behavior can also have an *initialization* and a *finalization* function. The agent will call the initialization function when it first starts executing the behavior and the finalization function when it finishes the execution, returning to the parent behavior or making the transition to the next behavior.

While making a transition, either decompositional or sequential, if there is more than one option, the agent will have to make the choice between the available behaviors. Thus each behavior will be associated with one decision making mechanism called *resolution* for each transition it has to make. The designer can either define a resolution method using HLBL or use one such as random or alternating choices provided by B2C. Each resolution is essentially an ordered set of mappings from condition sets to choices. When the agent tries to make a decision, the first set of conditions satisfied will result in the associated choice being the result. For an example, from B1, the agent can make a sequential transition to either B5 or B6. Based on the conditions C5 and C6, the agent will decide which behavior it should make a transition to.

In Figure 2.2, we show the pseudocode for the agent to execute the behavior hierarchy.

```
1. B ← Root of the behavior hierarchy
2. Repeat until an ending condition of the root is true:
3.         Initialize (B)
4.         Repeat until an ending condition of B is true
5.                 Perform the action of B
6.                 If B has a child C available for transition,
7.                         B ← C
8.                         Go to 2
9.         Finalize (B)
10.        If B has a following F available for transition,
11.                B ← F
12.        Otherwise B ← parent of B
```

Figure 2.2: Pseudocode for the execution of the behavior hierarchy.

2.2 Example of the Representation and Execution

In this section we will give an example of using the specified behavior hierarchy representation in implementing a program for the AIBO and how the program will be executed during run-time. This is part of the implementation we have done to evaluate the HLBL and the B2C system.

The AIBO is supposed to have the following behaviors:

- When it is placed on the ground, it will walk, trot, or run until it is lifted from the ground.
- If it is lifted straight up, it will wave its tail. If it is tilted to the left, it will turn on its middle left LED; if to the right, it will turn on its middle right LED.

Following is the correspondent behavior hierarchy:

The AIBO will start its execution from the ”Lifted Straight Behavior”. It will perform the action ”Waving Tail” as long as the AIBO is in this behavior. There are two sequential transitions from this behavior: to ”Tilted Behavior” when the AIBO is tilted, and to ”On Ground Behavior” when the AIBO is actually on ground. Thus the set of ending conditions of this behavior is composed of the conditions ”Is lifted” and ”Is back on ground”.

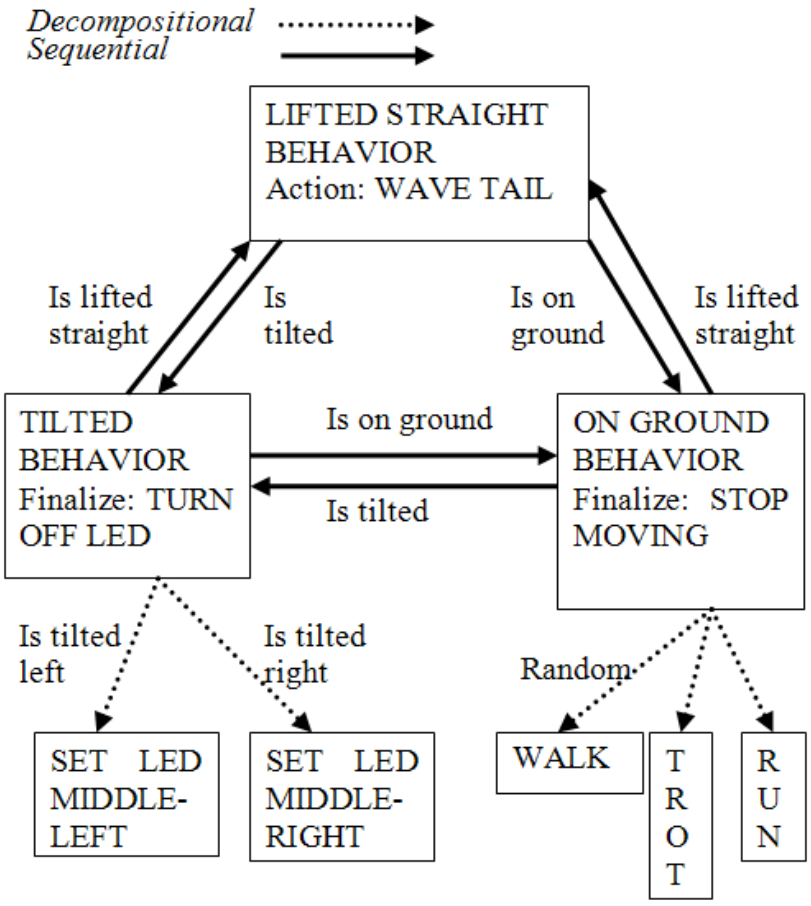

Figure 2.3: Behavior hierarchy for the AIBO.

If the AIBO makes a transition to the ”On Ground Behavior”, we can notice that the behavior has three decompositional transitions beside the two sequential transitions. The two sequential transitions are: to ”Tilted Behavior” when the AIBO is tilted, and to ”Lifted Straight Behavior” when the AIBO is lifted up straight. The ”On Ground Behavior” has three children: ”Walk”, ”Trot”, and ”Run”. When the AIBO starts executing ”On Ground Behavior”, since there is no action associated with this behavior, it will immediately make a transition to a random one among the behavior's children. Those children are atomic execution behaviors of the AIBO in the world. The ”On Ground Behavior” also has a finalization function ”Stop Moving.” Thus it will call this function once the AIBO stops executing the behavior. The set of ending conditions is composed of the conditions ”is lifted” and ”is back on ground”.

Similarly, the ”Tilted Behavior” has two decompositional transitions: to ”Set LED-Middle-Left” if the AIBO is tilted left and ”Set LED-Middle-Right” if the AIBO is tilted right. Based on the condition the AIBO will make the appropriate transition. Since ”Tilted Behavior” has a finalization function ”Turn off LED”, the AIBO will execute this function once it finishes executing this behavior. The two sequential transitions are: to ”On Ground Behavior” when the AIBO is actually on the ground, and to ”Lifted Straight Behavior” when the AIBO is lifted up straight.

2.3 HLBL Syntax and Specification

The language was designed to be a direct representation of the formal structural features of the agent as described in the previous section. The fields and keywords available are as follows:

- **behavior** <behavior−name> The behavior keyword specifies that the following information from this point until an end keyword is used to describe a behavior
- **startswhen** <condition> The startswhen keyword is followed by a condition that uses a Boolean expression composed from the condition functions. This expression is called the applicability condition.
- **endswhen** <condition> The endswhen keyword is similar to the startswhen keyword, but specifies the conditions upon which a particular behavior should cease execution.

- **children** <child 1> <child 2>...<child n> The children keyword describes the list of child behaviors reachable from the current behavior. These child behaviors correspond to a decompositional transition from the current behavior. This keyword is optional and if omitted, there is only one way to accomplish the behavior

- **following** <following1><following2>...<following> The following keyword specifies the next same-level behavior that should be executed after the current behavior. This keyword denotes sequential transition and it is optional.

- **child−resolution** <resolution−name> The child−resolution keyword specifies the decision making method that should be run at the current behavior, when deciding which of its alternative child decompositions should be taken. This keyword is present if and only if the behavior has some child.

- **following−resolution** <resolution−name> The follwing−resolution keyword has similar meaning as the child−resolution keyword except that it is dedicated for sequential transitions only.

- **action** <action−name> The action keyword is optional for behaviors with children but mandatory for the ones without. It specifies the atomic execution behavior that the agent needs to carry out within the current sense-plan-act cycle.

- **initialize** <action name> The initialize keyword is optional. It specifies the action that the agent needs to carry out once when it starts executing the behavior. This keyword is useful for resetting some internal state of the agent

- **finalize** <action name> The finalize keyword is similar with the initialize keyword. It is optional and can be used to specified the action that the agent needs to carry out once when it finishes executing the behavior.

- **resolution** <resolution name> The resolution keyword specifies that the following information until an end keyword is used to describe a method for the agent to make the choice among the behaviors that can be transitioned to.

- **cond** <condition> The cond keyword denotes the condition for one of the mappings in the behavior. It must be followed by a choice keyword

- **choice** <behavior name> The choice keyword follows a cond keyword to specify the behavior must be chosen if the followed conditions evaluated to true.

2.4 Example of Behavior Description in HLBL

```
behavior Lifted_Straight
        endswhen is_tilted || is_on_ground
        following Tilted On_Ground
        following_resolution state_based
        action Wave_Tail
end
behavior On_Ground
        endswhen is_tilted || is_up_straight
        following Lifted_Straight Tilted
        following_resolution state_based
        children Walk Trot Run
        child_resolution RANDOM
        finalize Stop_Moving
end
resolution state_based
        cond is_tilted
        choice Tilted
        cond is_on_ground
        choice On_Ground
        cond is_up_straight
        choice Lift_straight
end
```

Figure 2.4: The HLBL code of the AIBO

In Figure 2.4 we are showing part of the code for the behavior architecture of the AIBO with the behaviors described in Section 2.2 and Figure 2.3. There are two behaviors and one resolution function mentioned here: **Lifted_Straight** behavior, **On_Ground** behavior, and **state_based** resolution function. As described

in Section 2.2 and Figure 2.3, the AIBO will stop executing **Lifted_Straight** behavior when either it is tilted or it gets back on the ground. Thus the *endswhen* field contains the strings "is_tilted" and "is_on_ground". These strings denotes the condition checking functions in the platform-dependent library that need to be checked for the ending condition of the behavior.

When the AIBO stops executing **Lifted_Straight** behavior, it will make a transition to either **Tilted** or **On_Ground** behavior based on its current state. The names of those two behaviors appear in the *following* field of the behavior. The keyword *following_resolution* denotes the **state_based** function that will be used to decide which behavior the AIBO will make the transition to. The **state_based** function simply maps the current state of the AIBO to the appropriate behavior: if the AIBO is tilted, it will make the transition to **Tilted**, else if it is on the ground, it will make the transition to **On_Ground**. The **Lifted_Straight** also has an action field which is **Wave_Tail**. This action is an atomic execution function that is included in the platform-dependent library.

Similarly, the **On_Ground** behavior also has the *endswhen*, *following*, and *following_resolution* fields with the same meaning as in **Lifted_Straight** behavior. However it does not have an associated action but three children instead. Thus the *action* field is missing and there are the extra *children* and *child_resolution* fields. The three children of **On_Ground** are **Walk, Trot,** and **Run** as specified in the *children* field. These three behaviors are atomic execution functions in the platform-dependent library. The string RANDOM following the *child_resolution* field indicates that the AIBO will use the built-in random resolution method to decide which child it should execute. The AIBO will stop moving once it finishes the **On_Ground** behavior as specified by the *finalize* field.

2.5 Summary

HLBL is an intuitive and flexible platform-dependent scripting language that allows the designer to easily develop behavior-based architecture for agent system in the form of augmented behavior hierarchy. It increases reusability of code across different platforms since one particular architecture can be reused in similar solutions for other problems. HLBL also allows designer to easily modify the behavior architecture by tweaking the behavior hierarchy, the applicability conditions, and resolution methods in finding the optimal solution. Thus it can greatly reduce the developing and experimenting time and effort for the designers.

Chapter 3

Automated Code Generation

For an existing platform, there is a platform-dependent library of condition checking functions and atomic execution behaviors. Given the API of the provided library, the designer will specify the agent's behavior architecture in a behavior description file using the HLBL.

The B2C system will then translate this file into correspondent C++ code. Once compiled, this code will yield an executable that can perform the specified behavior control routines.

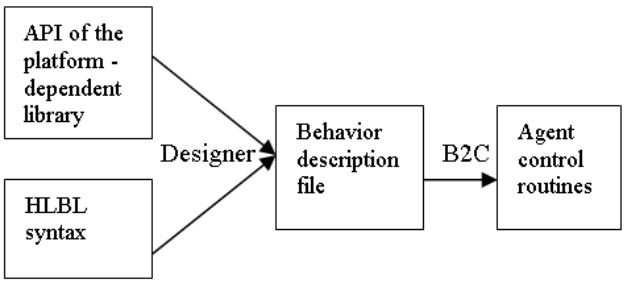

Figure 3.1: The process of generating agent control routines.

3.1 Platform-dependent Library

The library is composed of a set of condition−checking functions and a set of behavior execution functions. The condition checking functions are boolean func-

tions without arguments which compute some function of the current world state as perceived by the agent. As an example, the AIBO has sensors to detect if it is on the ground or not. Thus it has the boolean function is_on_ground which will return true if the AIBO has its feet on the ground and false otherwise.

The set of behavior execution functions, which can be called at the end of each perception−thought−action cycle of the agent, support the actual execution of the agent in the world. These functions return nothing and also take in no arguments. For example, a behavior might have an execution function ”Walk()” and thus as long as that behavior is active, the Walk() function will be called at every thinking cycle. The actual interior workings of the Walk() function are platform−dependent and as the agent merely calls the function Walk(), the actual execution of the function may involve sockets, threading operations, or any other code necessary to actuate the agent in the world. The library also includes a set of atomic execution functions which are the lowest level behaviors and not composed of any other behaviors.

3.2 Auto-translation of the Behavior Description to Code

There are two similar models available. They only differ in the control flow of the programs.

3.2.1 Implicit Continuous Control Model

B2C will take the behavior description file as input and generate a .cpp file with all the necessary links. Each behavior will be translated to a C++ function that has no argument and returns no value. A behavior makes transition to another one through the correspondent function call. The function begins with an if−statement that will terminate the function when the starting condition is not met. The body of the function is a while loop that will run until the ending condition is met. Inside the while loop, the action of the behavior will be called first and then the resolution function to get the name of the child for transition. The agent will call the function of the chosen child behavior if there is any.

After the while loop, the finalization function will be activated. The agent will then call the resolution function to choose one of the following functions. Upon receiving the result, the agent will call the function of the chosen following

behavior if there is any; otherwise it will return to the parent.

As indicated in previous sections, the set of ending conditions of a behavior include the ending conditions of its ancestor. Therefore B2C will recursively propagate the ending conditions of one behavior to its children until no new condition is added to a behavior. This guarantees each behavior will inherit all of its ancestor's ending conditions. If there is a loop of behaviors, the program will terminate when all the behaviors in the group have the same set of ending conditions.

The resolution method will be simply translated to a function that takes in no argument and returns the name of the chosen behavior. The function is composed of several if−statement in the same order as the condition−choice pairs described by the designer. Each statement will return a string when its if−condition is met. At the end of the function, if no condition is evaluated to true, the function will return an empty string.

In this model, the stack of the behaviors that the agent currently executes is implicitly maintained by the function-call stack of the compiler. Once the agent starts executing the behavior control routines, the main control flow of the program will remain inside the behavior control routines until the agent finishes the root behavior. Thus if the agent needs to process any perceptions of the world or perform any action in the world, it must possess another thread running in parallel with the behavior control routines thread, or the perceptions and the actions must be fast not to block the execution of the behaviors. The agent will carry out the Planning cycle continuously based on the conditions of the agent and the world.

In Figure 3.2 we show the native code version of part of the behavior architecture generated by B2C. The behaviors are described in Figure 2.3 and the correspondent behavior description file written in HLBL code is provided in Figure 2.4. The **call_function** function is a helper routine that will call the appropriate function based on the names of the function.

3.2.2 Explicit Discrete Control Model

In this model, each behavior is translated to a function in a similar fashion as in the previous model. The only difference is that there is an explicit behavior stacks for the behaviors the agent is executing. The behavior at the top of the stack is the active one. For every Planning cycle, the agent will check the appropriate conditions first and then either perform the active behavior or make a transition to another one. If the next behavior is the parent of the current one, the agent will just need to pop the current behavior off the stack; otherwise it will also have to push the new behavior to the stack. With the explicit stack, the main control

```
void call_function (string name) {
    if (name=="Lifted_Straight") Lifted_Straight ();
    if (name=="On_Ground") On_Ground ();
    if (name=="Tilted") Tilted ();
}
string state_based () {
    if (is_tilted()) return "Tilted";
    if (is_on_ground()) return "On_Ground";
    if (is_up_straight()) return "Lift_straight";
}
void On_Ground () {
    do {
        if (is_tilted()||is_up_straight()) break;
        string child=  RAND_On_Ground_Res();
        call_function (child);
    } while (true);
     Stop_Moving();
    string next=  state_based();
    if (next!="") {
        call_function (next);
        return;
    };
}
```

Figure 3.2: An excerpt from the translated code in Implicit Continuous Control Model.

flow of the program can be returned to an outer caller. Thus the agent can carry out the Planning cycle step by step and it does not have to invoke planning unless necessary.

In Figure 3.3 we show the native code for the same behaviors as in Figure 3.2 but with the Explicit Discrete Control Model. Note the main difference is in the helper function **call_function** and **operate**. The function **call_function** uses a stack and a counter to keep track of the behaviors and make the trasitions. The function **operate** calls the behavior at the top of the stack everytime it gets executed.

```
void call_function (string name) {
    beh_initialized=false;
    if (name=="Lifted_Straight") beh_stack[beh_counter]=0;
    if (name=="On_Ground") beh_stack[beh_counter]=2;
    if (name=="Tilted") beh_stack[beh_counter]=1;
    if (name=="Walk_Beh") beh_stack[beh_counter]=3;
    beh_counter++;
}
void On_Ground () {
    if (is_tilted()||is_up_straight()) {
        if (beh_initialized) Stop_Moving();
        beh_counter--;
        string next=  state_based();
        call_function (next);
        return;
    };
    if (!beh_initialized) {
        beh_initialized=true;
    };
    string child=  RAND_On_Ground_Res();
    call_function (child);
}
void operate() (string name) {
    if (beh_counter>=1) {
        if (beh_stack[beh_counter-1]==0) Lifted_Straight();
        if (beh_stack[beh_counter-1]==2) On_Ground();
        if (beh_stack[beh_counter-1]==1) Tilted();
        if (beh_stack[beh_counter-1]==3) Walk_Beh();
    };
```

Figure 3.3: An excerpt from the translated code in Explicit Discrete Control Model.

3.3 Experimentation with Different Behavior Architectures

To change the behavior hierarchy, the designer can simply edit the behavior description file. Once translated again, the new code will reflect the changes in the behaviors. There are several ways that a designer can tweak the behavior architecture such as changing the links between behaviors, the starting and ending conditions, the action associated with the behavior, or the resolution methods. These result in a wide range of performance of the agents. Since all the changes are

made in HLBL, the designer can spend little time and effort to experiment with different agent architecture. In Figure 3.4, we show an example of the changes in the native code correspondent with the changes in the behavior description file.

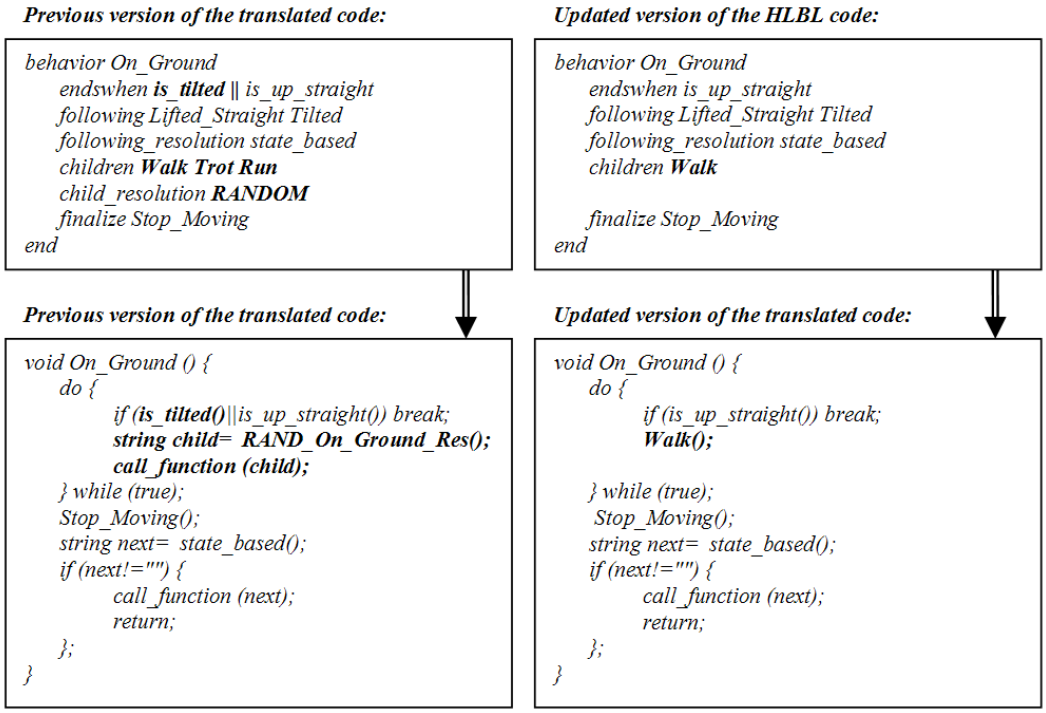

Figure 3.4: An example of how the translation reflects the changes of the behavior.

3.4 Summary

Given the platform-dependent library and the behavior description file, the B2C system will automatically generate native code correspondent to the desired behavior control routines. There are two control models that an agent designer can choose from: explicit and implicit control model, based on whether the designer wants to have control over each planning step or not. B2C allows a designer to develop a behavior architecture and easily experiment with different parameters of the architecture using HLBL instead of native code. The translation of HLBL code to native code needs to happen only once.

Chapter 4

Application of HLBL and B2C

HLBL and B2C have been applied in several scenarios. One of them is implementing behavior control routines for the AIBO. An example of the behaviors was shown in Figure 2.3. An excerpt of the correspondent behavior description file written in HLBL and the translated C++ code were shown in Figure 2.4 and Figure 3.2, respectively. Beside the program for AIBO, we have also developed two very different simulations utilizing HLBL and B2C: the Maze Game and the Space Game.

4.1 The Maze Game

The first simulation is a Maze Game in which the agent has to find the path from one given square in a maze to a target square. The agent can only move from one square to another side adjacent square when there is no wall between the two squares. The platform-dependent library is composed of functions for condition checking and atomic execution behavior such as checking if there is a wall on a particular side of the agent or moving the agent to an adjacent square in a particular direction. The game uses a very simple command line. There is one thread for the agent's behavior control routines and one for updating the world state, and the agent interacts with the world through appropriate function calls. The world state evolves in discrete time steps. Each step is defined by a movement of the agent.

Figure 4.1 show a complete run of the agent finding a path from the given square to its destination. The dotted lines represent the walls and the star symbol represents the agent. There are 18 steps in total and the order of the steps is from the top to the bottom first and then from the left to the right. The main part of the

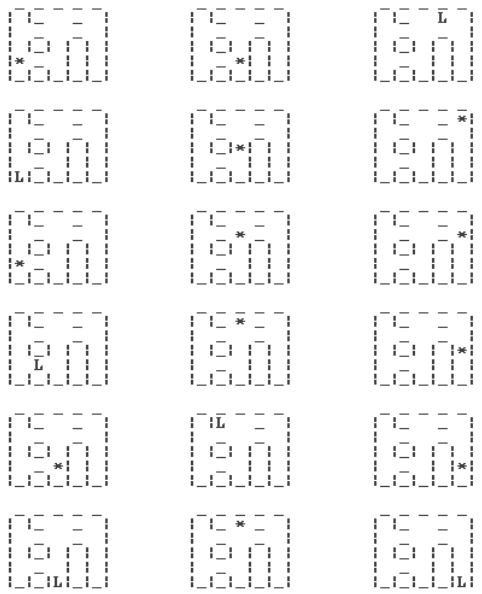

Figure 4.1: A screenshot of the Maze Game.

HLBL code for the agent is shown in Figure 4.2. The agent will attempt moving towards the target using the shortest path and backtracking if necessarily. The memorization of visited squares is done as part of the platform-dependent library.

4.2 The Space Game

The second simulation is a Space Game in which the agent acts as a space ship trying to break down oncoming asteroids. In contrast to the other game, this game has a more complicated Direct3D display as in Figure 4.3 and runs as a continuous-time simulation. The world states evolves without waiting for an action from the agent. The space, however, is discretized into squares such that the agent can only move from one square to another side adjacent square. The asteroids will move in on straight lines perpendicular with the surface that contains the agent. The agent can only shoot at the asteroids once it is aligned with them. To interact with the world, the agent uses sockets instead of function calls as in the Maze Game. The platform independent library includes functions and behaviors such as checking if there is an asteroid coming close by or moving one step in a particular direction.

```
behavior SearchMaze
        endswhen IsAtGoal
        children GoVertically GoHorizontally
        child_resolution SearchMazeSplit
end
behavior GoVertically
        endswhen !ShouldGoVertically
        children MoveDown MoveUp
        child_resolution GoVerticallySplit
end
behavior MoveDown
        endswhen !CanWalkDown
        action WalkDown
end
behavior MoveUp
        endswhen !CanWalkUp
        action WalkUp
end
behavior GoHorizontally
        endswhen !ShouldGoHorizontally
        children MoveLeft MoveRight
        child_resolution GoHorizontallySplit
end
behavior MoveLeft
        endswhen !CanWalkLeft
        action WalkLeft
end
behavior MoveRight
        endswhen !CanWalkRight
        action WalkRight
end
```

Figure 4.2: Part of the code of the agent for the Maze Game.

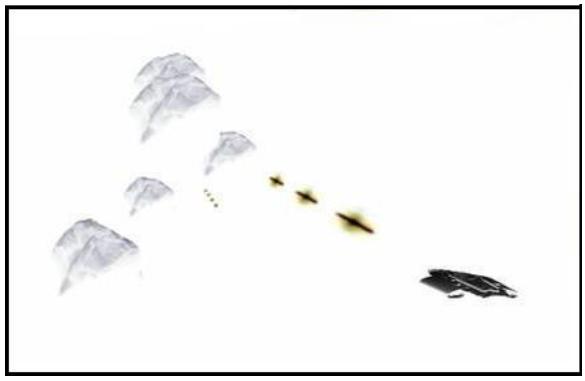

Figure 4.3: A screenshot of the Space Game.

The code for the agent is shown in Figure 4.4. The agent will attemp to move to the closest square that is aligned to an oncoming asteroid. The behaviors **GoVertically** and **GoHorizontally** allows the agent to move to the desired position. Once it have an asteroid in range as indicated by **HasTargetStraight** function, it will stop moving and start shooting at the asteroid using **Shoot** action. Both of the condition checking function and the atomic action are provided from the platform-dependent library.

4.3 Advantages of Using HLBL and B2C

The agents in both games were running using compiled C++ code automatically translated from behavior description file in HLBL. HLBL significantly reduced the time required to implement the behavior control routines for these agents, and B2C allowed rapid integration of the routines into the underlying platform. Moreover, even though these two games are very different, the agents in these games still share a large part of their architectures, especially evident in the top-level behaviors. An example is given in Figure 4.5. The agents in both games have very similar behavior ”GoVertically”. The only difference is their ending conditions. Thus, we were able to reuse several parts from the behavior architecture of the agent in the Maze Game when implementing the agent for the Space Game. This further facilitated the process of developing the agent.

With the behavior architecture implemented in HLBL, it is very easy to tweak different parameters such as the dependency between behaviors, the applicability

```
behavior ShootAsteroid
        endswhen TimeIsOver
        children KeepPosition GoVertically GoHorizontally
        children_resolution ShootAsteroidSplit
end
behavior KeepPosition
        endswhen !HasTargetStraight
        execution Shoot
end
behavior GoVertically
        endswhen !ShouldGoVertically —— HasTargetStraight
        children WalkDown WalkUp
        children_resolution GoVerticallySplit
        following KeepPosition
end
behavior WalkDown
        endswhen !ShouldGoDown
        execution MoveDown
end
behavior WalkUp
        endswhen !ShouldGoUp
        execution MoveUp
end
behavior GoHorizontally
        endswhen !ShouldGoHorizontally —— HasTargetStraight
        children WalkLeft WalkRight end
        children_resolution GoHorizontallySplit
        following KeepPosition
end
behavior WalkLeft
        endswhen !ShouldGoLeft
        execution MoveLeft
end
behavior WalkRight
        endswhen !ShouldGoRight
        execution MoveRight
end
```

Figure 4.4: Part of the code of the agent for the Space Game.

and ending conditions, or the resolution methods. We could focus on programming the agents at the abstract behavior level without being burdened with the implementation details. Therefore we were able to experiment with many different behavior architectures in a short time in the pursuit of the optimal one. Our experience with using HLBL and B2C as described above demonstrates the advantages of using the framework to implement and experiment with agent behavior architectures.

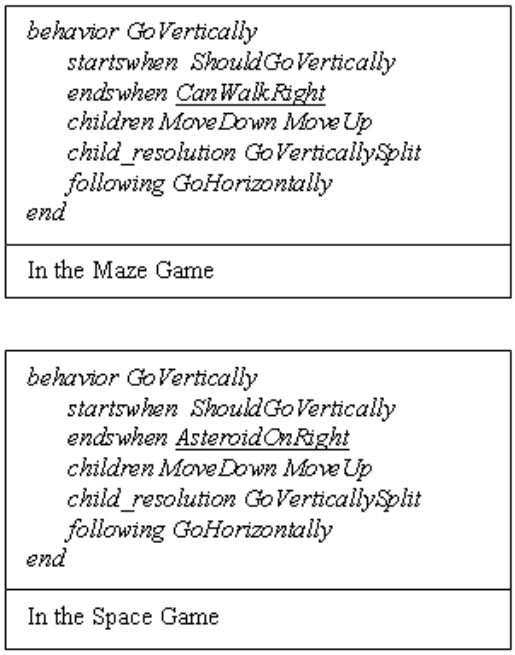

Figure 4.5: An example of reusing behaviors.

4.4 Summary

We have applied HLBL and B2C in developing agent behavior controls for the AIBO on the existing platform of CMRoboBits, and for two simulated environment Maze Game and Space Game. The successful implementation of several agent behavior architectures with different goals and structures on different platforms demonstrates the capability of HLBL to represent a wide range of behavior

architectures including finite state machines. It also demonstrates the capability of B2C to generate native code from HLBL code of the behaviors and integrate that code to different platforms including software and hardware ones. Using HLBL and B2C, we have experienced several advantages such as more rapid developing and experimenting process, and better reuse of the implemented behavior architecture.

Chapter 5

Conclusion

The main contribution of this thesis is a novel framework for agent programming with two main components HLBL and B2C. HLBL is a platform-independent language with which one can design modular behavior architectures at the abstract behavior definition level. With this intuitive and flexible language, a designer can make rapid modifications to an agent's behavior architecture for experimentation and testing purposes. The automated code generation component, B2C, further facilitates the process of developing and experimenting with different behavior architectures by allowing designers to quickly integrate complex HLBL hierarchies directly into their agent's code. HLBL and B2C also help increasing reusability of agent architectures across platforms as part of an implemented architecture can be easily reused on another platform. We have implemented several agent behavior architectures on a variety of platforms to demonstrate these advantages of using HLBL and B2C. Possible extension of the framework may include developing a graphic interface for HLBL to further facilitate the designing process and expanding HLBL and B2C to allow an agent to execute more than one behavior in parallel and to support multi-agent systems.

Bibliography

[1] A. Brooks. A Robust Layered Control System for a Mobile Robot. In *IEEE Journal of Robotics and Automation RA-2 (1)*, pages 14–23, 1986.

[2] E. Gat. ALFA: A Language for Programming Reactive Robotic Control Systems. In *IEEE Conference on Robotics and Automation*, 1991.

[3] G. Kaminka, J. Go, and T. Vu. Context-Dependent Joint-Decision Arbitration for Computer Games. In *Proceedings of the Agent In Computer Games Workshop at ICCG'02*, 2002.

[4] S. Lenser, J. Bruce, and M. Veloso. CMPack: A Complete Software System For Autonomous Legged Soccer Robots. In *Proceedings of the Fifth International Conference on Autonomous Agents*, 2001.

[5] M. Veloso, S. Lenser, D. Vail, P. E. Rybski, N. Aiwazian, and S. Chernova. CMRoboBits: Creating an Intelligent AIBO Robot class. In *Proceedings of the AAAI Spring Symposium on Accessible Hands-on Artificial Intelligence and Robotics Education*, March 2004.

[6] M. Veloso, D. Vail, S. Lenser, S. Chernova, J. Bruce, M. Roth, and J. Fasola. CMPack'03: Robust Modeling and Coherent Teamwork. 2003.

[7] T. Vu, J. Go, G. Kaminka, M. Veloso, and B. Browning. MONAD: A Flexible Architecture For Multi-Agent Control. In *Proceedings of the AAMAS'03*, pages 449–456, 2003.